# 流量优化

## 接入优化

移动互联网下，网络不稳定，而且经常会出现运营商切换，dns劫持等状态导致移动端设备接入失败，针对这个情况，需要有一套流程来保障接入的体验优化：

Todo:加入对应活动图

## 状态推送

移动互联网掉线离线太频繁，没有必要把Presence状态公布给好友的；还有群，没有必要把所有在线状态的变化通知给群里面所有人，要实现类似微信的聊天室，不在线的时候把消息推给他。

## 消息推送

方案对比：

|  |  |  |
| --- | --- | --- |
| 方案 | 优点 | 缺点 |
| 全推 | 实现简单 | 耗流量，比较适合有线网络  多终端时容易出现漏 |
| 2G、3G下仅推送消息编号 | 节省流量  多终端不会出现漏 | 2G、3G下无法看到消息预览 |

## 消息格式

方案对比：

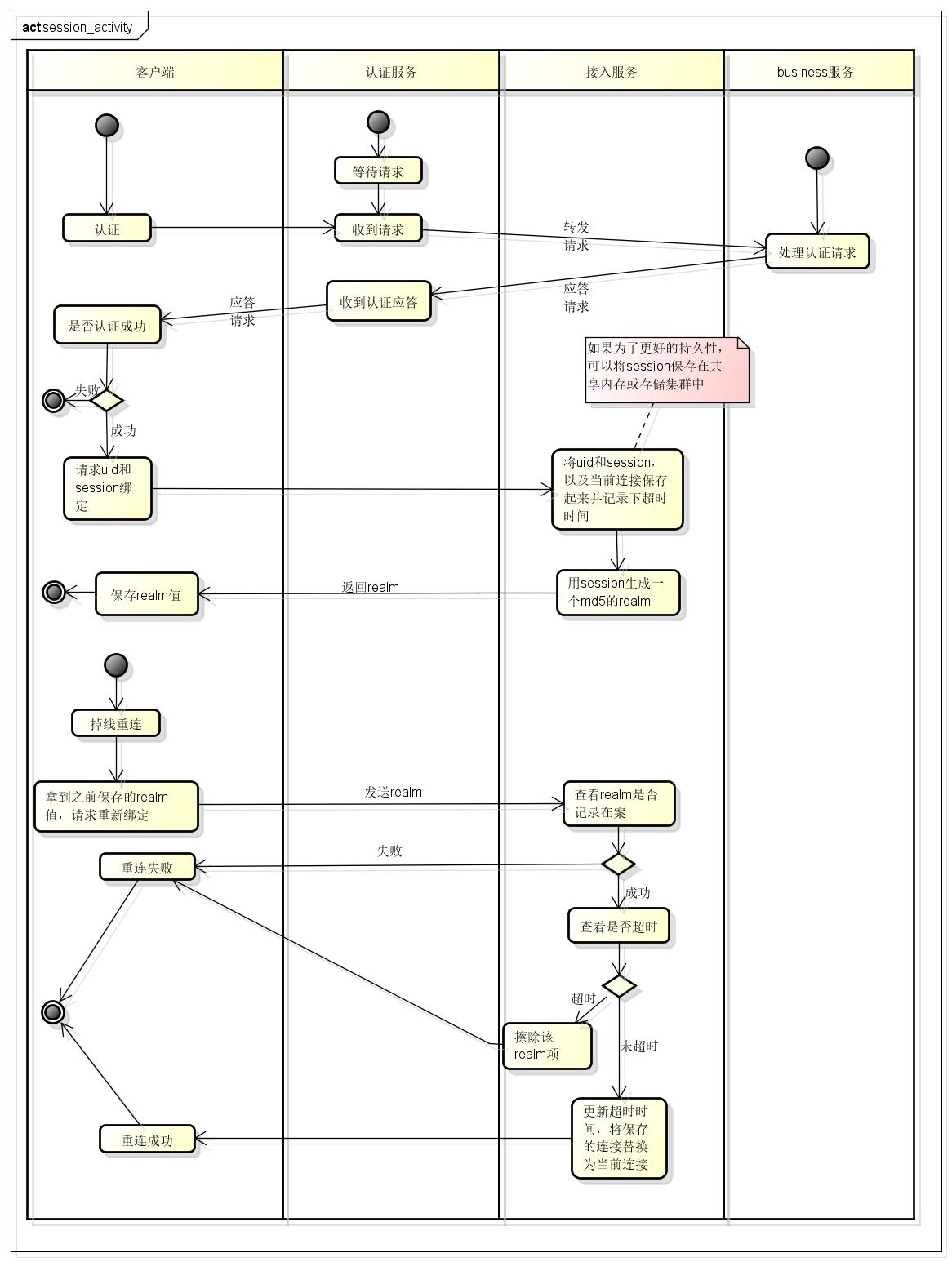
|  |  |  |
| --- | --- | --- |
| 方案 | 优点 | 缺点 |
| Protobuf | 1. 接口灵活，可以动态添加字段，预留字段 2. 压缩率高，比xml小3~10倍 3. 编解码高效，比xml快20~100倍 | 1. 应用不够广，难以跟第三方对接 2. 二进制格式导致可读性差 3. 缺乏自描述，如果缺乏proto文件，无法知晓内容 |
| json，xml | 可读性高，易编辑 | 占空间和流量，且效率不高 |

注：更详细的是数据对比已经在《共享平台移动IM服务设计》中有体现，这里不再赘述。

## 掉线重连

方案对比：

|  |  |  |
| --- | --- | --- |
| 方案 | 优点 | 缺点 |
| Session，掉线后客户端可以根据session来自动重连 | 高效，用户体验好，如果软件没有在前台，用户几乎感受不到掉线上线的动作 | 服务端要共享状态，对性能有点影响 |
| 无session，掉线后客户端要重新认证 | 服务端简单 | 客户端需要做大量工作  效率低下，用户体验差 |



## 消息补齐

方案对比：

|  |  |  |
| --- | --- | --- |
| 方案 | 优点 | 缺点 |
| 使用消息ID来补齐缺失的消息 | 1. 类似svn操作，根据客户端和服务端的消息ID差值来接收消息，不会产生消息空洞。 2. 只同步增量，节省流量 3. 一次可以发送多条消息 | 需要一个稳定的全局ID生成器 |
| 服务端批量下发全部消息 | 客户端操作简单，只需等待服务端推送 | 消耗大量流量，移动端无法支撑 |

具体通过消息ID来进行消息推送的逻辑在《消息时序》中有体现，这里不再赘述。

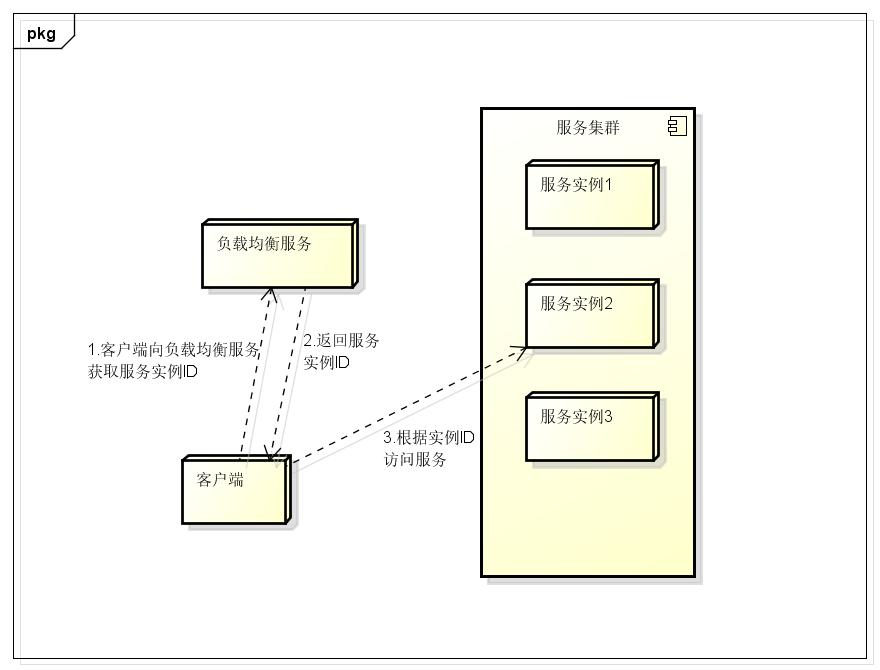
# 动态扩容

假设一组机器来提供一个服务，客户端要以相同的机会访问各台机器，而且其中一台机器负载过高的时候，要减少对这台服务器的访问，直到它的负载降低下来，而且如果添加了一台新的服务器，要把客户端的请求也均衡到这台新机器上。

## 负载均衡服务

要将客户端的访问均衡地分配到每个服务器，需要有一个额外的负载均衡服务器，需考虑以下几点：

1. 负载均衡服务器本身是个单点，如果它本身也要支持流量负载的话，要把它本身的状态放到数据库里。
2. 在某台机器宕机的时候把请求路由到其它机器。
3. 负载均衡的算法必须要均匀分布。

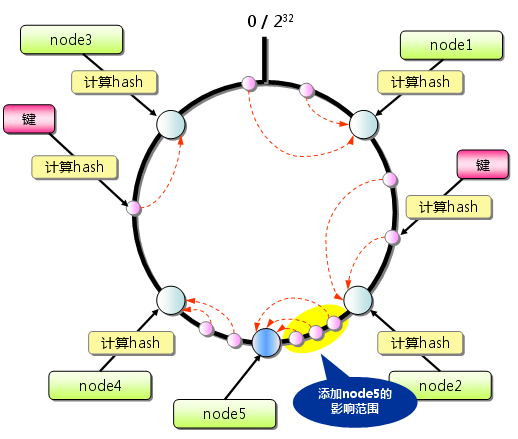


## Hash算法

方案对比：

|  |  |  |
| --- | --- | --- |
| 方案 | 优点 | 缺点 |
| 一致hash算法 | 服务增减只会导致少量hash失效 | 实现复杂 |
| 普通hash算法 | 实现简单 | 服务增减只会导致绝大部分hash失效 |

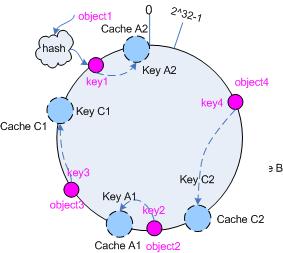
一致hash的算法描述图如下：



一致性hash的算法解释如下：

1. 将这个空间想象成一个首（ 0 ）尾（ 2^32-1 ）相接的圆环
2. 将对象通过算法hash(node) = key，将node映射到圆环上
3. 将客户端ID也通过hash算法映射到圆环上
4. 沿着客户端映射到圆环上的位置，顺时针查找，直到找到node节点，即为目的节点
5. 如果添加一个节点，例如上图中的node5，只会影响node2到node5之间的区间

另外，一致hash有个固有缺陷，就是每个节点管理的区域可能不均匀，因此，可以引入虚拟节点解决这个问题，虚拟节点描述如下：



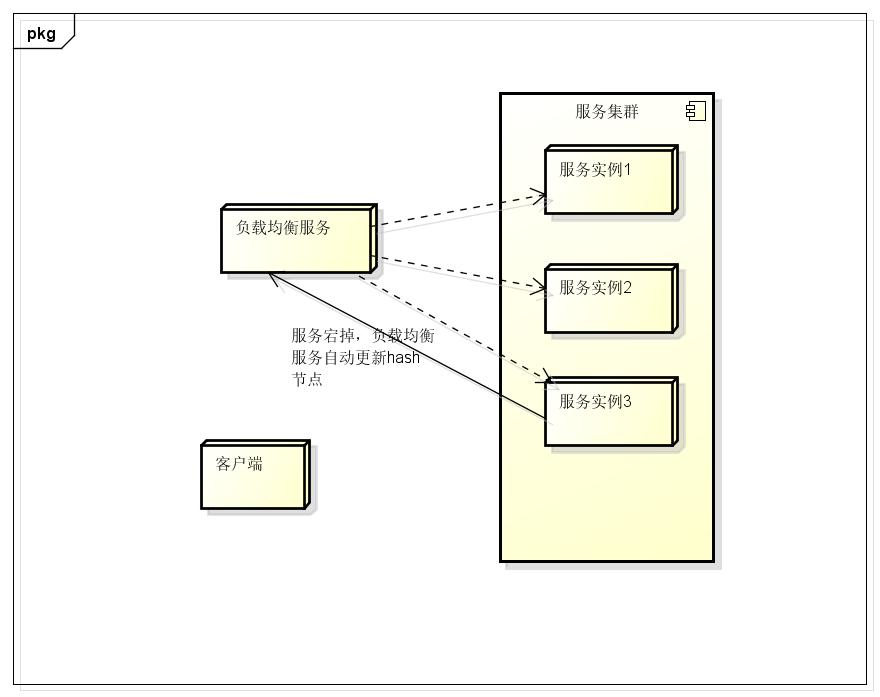
虚拟节点的描述如下：

1. 将一个服务器按照权重映射成多个节点，如图中A映射成A1，A2，C映射成C1，C2
2. 将客户端ID通过hash算法映射到圆环上
3. 如果按顺时针查找，首先匹配到A1或A2，即为匹配服务器A，否则匹配服务器C

## 服务探测

方案对比：

|  |  |  |
| --- | --- | --- |
| 方案 | 优点 | 缺点 |
| 自动探测服务增减的事件，调整hash策略 | 减轻运维工作量 | N/A |
| 手动更新配置 | N/A | 加大运维工作量，更新不及时 |



## 服务状态

方案对比：

|  |  |  |
| --- | --- | --- |
| 方案 | 优点 | 缺点 |
| 服务无状态，核心数据放入集群共享存储 | 持久性最好，服务器重启，服务也可以继续工作 | 效率低下 |
| 服务无状态，核心数据放入共享内存 | 持久性中等，服务重启可以继续工作 | 服务器重启状态丢失 |
| 服务状态记录在内存中 | 效率最高 | 服务重启状态丢失 |

## 数据库

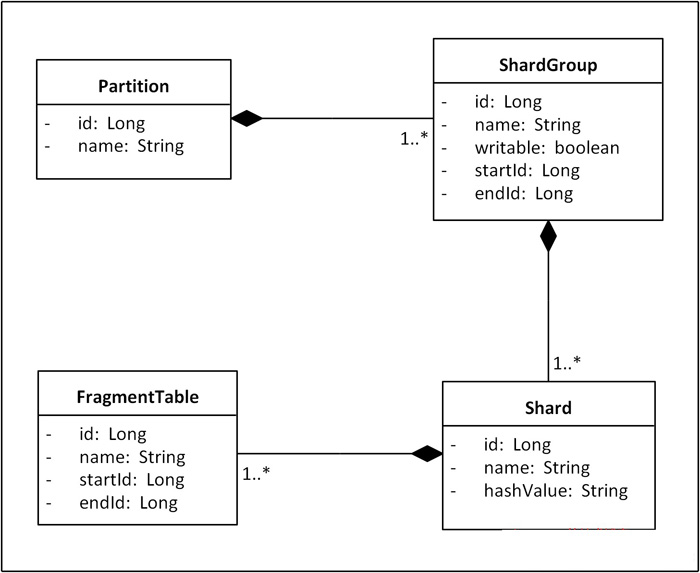
理想的数据库扩容应该努力满足以下几个要求：

1. 最好不迁移数据 （无论如何，数据迁移都是一个让团队压力山大的问题）
2. 允许根据硬件资源自由规划扩容规模和节点存储负载
3. 能均匀的分布数据读写，避免“热点”问题
4. 保证对已经达到存储上限的节点不再写入数据

针对这四个要求，首先引入几个概念：

1. 对单一的数据库做垂直切分，垂直切分将关系密切的表划分在一起，这样分出来的表称为partition。
2. 对partition内的表做水平切分，水平切分将一张表按增量区间或散列方式切分，分散到多个shard上。
3. 将散列的shad组合到一起，称为ShardGroup。
4. ShardGroup内的每个Shard创建相同的表，称为FragmentTable。

这四个概念的关系如下图所示：



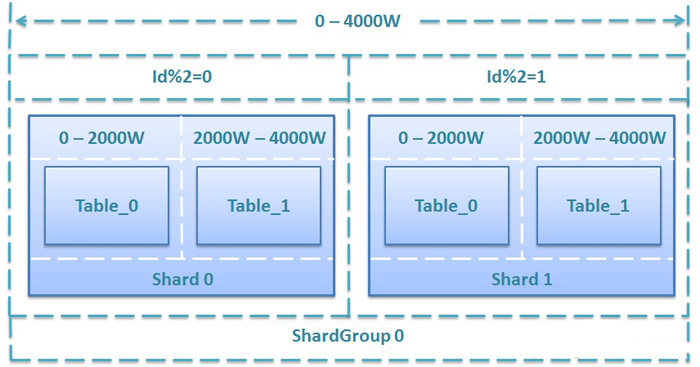
其中，ShardGroup解决数据迁移和容量增加的问题，新增的服务器为新添加的ShardGroup，无需迁移数据；FragmentTable解决热点的问题；ShardGroup的writable解决旧存储不再写的问题。

### 示例

让我们通过示例来了解这套方案是如何工作的：

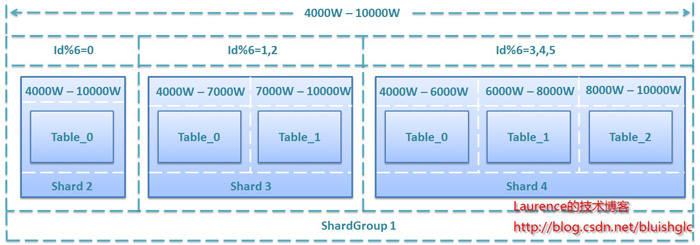
**阶段一：初始上线**

假设某系统初始上线，规划为某表提供4000W条记录的存储能力，若单表存储上限为1000W条，单库存储上限为2000W条，共需2个Shard，每个Shard包含两个分段表，ShardGroup增量区间为0-4000W，按2取余分散到2个Shard上，具体规划方案如下：



**阶段二：系统扩容**

经过一段时间的运行，当原表总数据逼近4000W条上限时，系统就需要扩容了。为了演示方案的灵活性，我们假设现在有三台服务器Shard2、Shard3、Shard4，其性能和存储能力表现依次为Shard2<Shard3<Shard4，我们安排Shard2储存1000W条记录，Shard3储存2000W条记录，Shard4储存3000W条记录，这样，该表的总存储能力将由扩容前的4000W条提升到10000W条，以下是详细的规划方案：



**阶段三：不扩容，重复利用再生存储空间**

假设系统又经过一段时间的运行之后，二次扩容的6000W条存储空间即将耗尽，但是由于系统自身的特点，早期的很多数据被删除，Shard0和Shard1又各自腾出了一半的存储空间，于是ShardGroup0总计有2000W条的存储空间可以重新利用。为此，我们重新将ShardGroup0标记为writable=true，并给它追加一段ID区间：10000W-12000W，进而得到如下规划方案：

